$\mathcal{Q}$WIRE: A Core Language for Quantum Circuits

Jennifer Paykin Robert Rand Steve Zdancewic
University of Pennsylvania, USA
jpaykin@seas.upenn.edu, rrand@seas.upenn.edu, stevez@cis.upenn.edu

Abstract

This paper introduces $\mathcal{Q}$WIRE ("choir"), a language for defining quantum circuits and an interface for manipulating them inside of an arbitrary classical host language. $\mathcal{Q}$WIRE is minimal—it contains only a few primitives—and sound with respect to the physical properties entailed by quantum mechanics. At the same time, $\mathcal{Q}$WIRE is expressive and highly modular due to its relationship with the host language, mirroring the QRAM model of computation that places a quantum computer (controlled by circuits) alongside a classical computer (controlled by the host language).

We present $\mathcal{Q}$WIRE along with its type system and operational semantics, which we prove is safe and strongly normalizing whenever the host language is. We give circuits a denotational semantics in terms of density matrices. Throughout, we investigate examples that demonstrate the expressive power of $\mathcal{Q}$WIRE, including extensions to the host language that (1) expose a general analysis framework for circuits, and (2) provide dependent types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

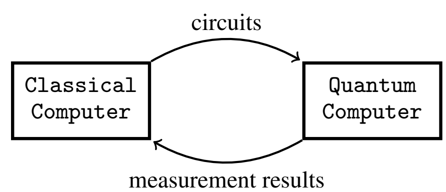
Keywords quantum programming languages, quantum circuits, linear types, denotational semantics

1. Introduction

The standard architecture for quantum computers follows the *quantum circuit model*, which presents quantum computations as sequences of gates over qubits (the quantum analogue of bits). As with classical circuits, quantum circuits exist at a very low level of abstraction, and yet in spite of this, researchers and industry professionals write complex quantum algorithms in state-of-the-art quantum circuit languages like Quipper (Green et al. 2013a) and LIQ$Ui|\rangle$ (Wecker and Svore 2014).

Why is the quantum circuit model so successful? In part, it is due to the fact that quantum data like qubits are extremely unintuitive from a classical perspective. Research into simple operations on quantum data, such as qubit-controlled conditionals and recursion, is still in its infancy (Ying 2014; Badescu and Panangaden 2015), so programmers cannot be sure that their algorithms using such abstractions are valid quantum-mechanically.

Although circuits manipulate quantum data, they themselves are classical data—a circuit is just a sequence of instructions describing how to apply gates to wires. In practice this means that circuits can be used to build up layers of abstractions hiding the low-level details. The QRAM model of quantum computing (Knill 1996) formalizes this intuition by describing how a quantum computer could work in tandem with a classical computer. In the QRAM model the classical computer handles the majority of ordinary tasks, while the quantum computer performs specialized quantum operations. To communicate, the classical computer sends instructions to the quantum machine in the form of quantum circuits. Over the course of execution, the quantum computer sends measurement results back to the classical computer as needed.



Embedded languages like Quipper, LIQ$Ui|\rangle$, the Q language (Bettelli et al. 2003), and the quantum IO monad (Altenkirch and Green 2010) can be thought of as instantiations of this model. They execute by running host language programs on the classical computer, making specialized calls to the (hypothetical) quantum machine. The classical host languages allow programmers to easily build up high-level abstractions out of low-level quantum operations.

However, such abstractions are only worthwhile if the circuits they produce are safe—if they do not cause errors when executed on a quantum computer. Unfortunately, proving that an embedded language produces well-formed circuits is hard because it means reasoning about the entirety of the classical host language. This is frustrating when we care most about the correctness of quantum programs, which we expect to be both more expensive and error-prone than the embedded language's classical programs.

One way of ensuring the safety of circuits is via a strong type system. Type safety for a quantum programming language means that well-formed circuits will not get stuck or "go wrong" when executed on a quantum machine. A subtlety is that this definition implies that the quantum program is even implementable on a quantum computer—that the high-level operational view of the language is compatible with quantum physics. One way of ensuring that the language is implementable is to give a denotational semantics for programs in terms of quantum mechanics.

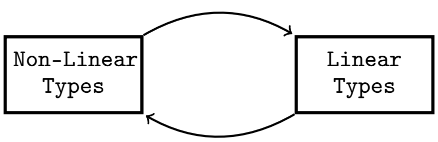
Several quantum programming languages have been proposed with an emphasis on type safety, including Selinger's QPL language (Selinger 2004), the quantum lambda calculus (Selinger and Valiron 2009), QML (Altenkirch and Grattage 2005), and Proto-Quipper (Ross 2015). However, these are toy languages, not designed for implementation in a conventional programming environment.

In this paper we address the tension between expressive embedded languages and denotationally-sound type-safe languages.

The best of both worlds: $\mathcal{Q}$WIRE

We propose the design of a core quantum circuit language in which circuits, equipped with a purely linear type system to ensure type safety, are explicitly separated from an arbitrary classical host language. The circuit language, which we call $\mathcal{Q}$WIRE ("choir"), comes equipped with an interface to this host language that allows for all the benefits of an embedded language while maintaining type safety and soundness.

The quantum lambda-calculus popularized the use of linear types for quantum systems. The "no-cloning" theorem of quantum mechanics states that quantum data cannot be cloned; in a programming environment, linear types ensure that quantum programs do not try to violate this property. However, the programming model should also allow for non-linear programming of ordinary classical data. The quantum lambda calculus addresses this via subtyping, but for $\mathcal{Q}$WIRE we take an alternative approach inspired by the symmetry between the QRAM model and Benton's Linear/Non-Linear (LNL) logic (1995):



In $\mathcal{Q}$WIRE, quantum circuits execute on the quantum computer and are given linear types, while host language programs execute on the classical computer and are given ordinary non-linear types.

Structuring the system in this way has several advantages. First, the interface to circuits is minimal, which means that they can be easily studied reasoned about. Second, the host language is extensible, since changes to the host language don't induce changes to the circuit language, and vice versa. Third, the relationship between the circuit language and host language can be easily axiomatized: every circuit can be promoted to the host language via a *box* operator, and then *unboxed* to be reused inside of other circuits. This allows circuits to be treated as classical data structures in the host language, while prohibiting quantum data such as qubits from escaping the linear type system.

The axiomatic approach means that the circuit language is relatively independent from the host language. In particular, we expect that the host language could be instantiated with a wide range of programming languages depending on the intended use: high-level functional programming languages for developing and reasoning about algorithms; theorem provers for verification of quantum circuits; and perhaps even hardware description languages for deployment with real quantum computers.

Contributions.

- We present $\mathcal{Q}$WIRE, a core quantum circuit language, along with a simple linear type system (Section 3) and an equational operational semantics (Section 4). In addition to the circuit language itself, we describe a minimal interface to a classical host language that allows for modularity and communication via the QRAM model.
- We prove that the operational semantics of $\mathcal{Q}$WIRE is type safe (Theorems 6 and 7), and that all circuits reduce to a small set of normal forms (Theorem 8), depending only on the correctness of the host language.
- We give a denotational semantics in terms of density matrices (Section 5) and prove that the operational semantics is sound with respect to it (Theorem 11).
- Throughout we give examples of circuits written in an archetypal host language with access to $\mathcal{Q}$WIRE (Section 2). We also consider how to extend the host language with case analysis of circuits and dependent types (Section 6) to express programs that cannot be written in existing circuit languages.

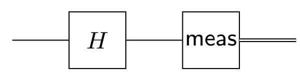
2. $\mathcal{Q}$WIRE by Example

We start by taking a look at some code written in a host language that has access to $\mathcal{Q}$WIRE circuits. Circuits are constructed by a *box* operator that binds the input, represented as a *wire name*, inside of a circuit. Each wire name is identified with a *wire type*, which is either a bit, a qubit, or a (possibly empty) product of wire types.

For example, the identity circuit is written `id` $=$ `box` $w \Rightarrow$ `output` w and has the type $\mathsf{Circ}(W, W)$ for any wire type W. The wire name w is not a regular variable as one would use in a classical programming language like the host language. For one, a wire is not first class: it is not by itself a circuit. For another, wire variables can only be used inside a circuit, and must be used *linearly*—once it is used, the same wire cannot be used again.

Gate application is the most important operation on wires. For example, the following circuit applies a Hadamard gate (`H`) to its input wire, followed by a measurement gate. Each gate has an associated input and output type and can only be applied to wires of the appropriate type.

```
hadamard-measure : Circ(qubit,bit) =
  box w =>
    w' <- gate H w;
    b  <- gate meas w';
    output b
```



Note that we sometimes write (`gate g w`) as shorthand for the (`w' <- gate g w; output w'`) that appears in the example.

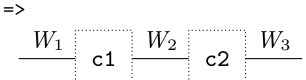
The reason wires must be treated linearly is that applying a gate changes the nature of the wire `w`. It is meaningless to apply two gates to the same wire, because wires (and in particular qubits) cannot be duplicated. The following code, for example, is absurd:

```
absurd = box w =>
  x  <- gate meas w;
  w' <- gate H w;
  output (x,w')
```

Similarly, it is dangerous to implicitly discard references to wires, which might be entangled in a greater quantum system. In $\mathcal{Q}$WIRE the `discard` gate explicitly discards a bit-valued wire, whereas qubit-valued wires must be measured before being discarded.

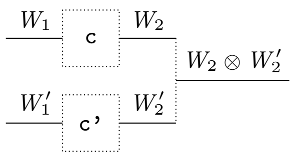
Since gates act on wires and not entire circuits, the expression `gate meas (gate H w)` is ill-formed. However, circuits can be composed by connecting the output of one circuit to the input of another. This type of composition is most useful when using circuits that have previously been constructed by a *box* operator. Boxed circuits can be *unboxed* by connecting some free input wires to the input of the box. The following function composes two boxed circuits in sequence, resulting in one complete circuit:

```
inSeq (c1 : Circ(W1,W2)) (c2 : Circ(W2,W3))
    : Circ(W1,W3) = box w1 =>
      w2 <- unbox c1 w1;
      unbox c2 w2
```



The type system guarantees that the output wire of the first circuit matches the input wire to the second. More complex composition is also possible. For instance, inPar composes any two circuits in parallel, with no restriction on their wire types.

```
inPar (c : Circ(W1,W2)) (c' : Circ(W1',W2')
  : Circ(W1⊗W1', W2⊗W2') =
  box (w1,w1') =>
    w2  <- unbox c  w1;
    w2' <- unbox c' w1';
    output (w2,w2')
```



In the host language, we can write functions that compute circuits based on classical values, such as the following initialization function for qubits that determines which initialization gate gets applied.

```
init (b : Bool) : Circ(1,qubit) =
  if b then box () => gate init1 ()
       else box () => gate init0 ()
```

Quantum teleportation. The quantum teleportation algorithm (adapted from Green et al.'s introduction to the Quipper language, 2013b) highlights the relationship between boxed and unboxed circuits more clearly. Figure 1 shows the quantum teleportation circuit, broken up into four parts. Alice is trying to send a qubit q, the input to the teleport circuit, to Bob. The circuit bell00 initializes two qubits in the zero state (written $|0\rangle$), places qubit a in a superposition of $|0\rangle$ and $|1\rangle$ via the Hadamard (H) gate, and entangles it with qubit b by applying a controlled-not (CNOT) gate. Qubit a is then given to Alice, and qubit b to Bob. Alice entangles a and q and measures them, outputting a pair of bits x and y. Bob then uses these to transform his own qubit into the state of the original qubit q.

Communication via lifting. In the teleportation example, the bit-valued wires x and y are treated as controls in the bob circuit. Intuitively, the bits x and y contain classical information, and so they should be able to be manipulated in by the host language. The *dynamic lifting* operation promotes bits to the host language so they can be manipulated using classical reasoning principles.[1] The bob circuit could be written instead using dynamic lifting:

```
bob-dyn : Circ(bit⊗bit⊗qubit, qubit) =
  box (w1,w2,q) =>
    (x1,x2) <= lift (w1,w2);
    q <- unbox (if x2 then X_gate else id) q;
    unbox (if x1 then Z_gate else id) q
```

where X_gate = box w => gate X w and similarly for Z_gate.

On the one hand, dynamic lifting produces legible code that is easy to understand because it concentrates more computation in the host language. On the other hand, dynamic lifting is inefficient because the host language code must be run on a classical computer, during which time the quantum computer must remain suspended, waiting for the remainder of the circuit to be computed. Although dynamic lifting is not necessary in the case of quantum teleportation, it is an integral part of many quantum algorithms including quantum error correction, and so must be accounted for coherently.

The examples shown so far describe all of the ways to construct circuits in $\mathcal{Q}$WIRE. However, when describing quantum algorithms, circuits are ultimately intended to be executed on a quantum computer. The final piece of the story is therefore the *run* operation, which takes a circuit with no input and produces a value. For example, the following code implements a quantum coin toss:

[1] Dynamic lifting can be applied to qubits as well as bits by implicitly measuring the qubit before producing a host-language value.

```
bell00 : Circ(1,qubit⊗qubit) =
  box () =>
    a <- gate init0 ();
    b <- gate init0 ();
    a <- gate H a;
    gate CNOT (a,b)

alice : Circ(qubit⊗qubit, bit⊗bit) =
  box (q,a) =>
    (q,a) <- gate CNOT on (q,a)
    q     <- gate H q;
    x     <- gate meas q;
    y     <- gate meas a
    output (x,y)

bob : Circ(bit⊗bit⊗qubit, qubit) =
  box (x,y,b) =>
    (y,b) <- gate (bit-control X) (y,b);
    (x,b) <- gate (bit-control Z) (x,b);
    ()    <- gate discard y;
    ()    <- gate discard x;
    output b

teleport : Circ(qubit,qubit) =
  box q =>
    (a,b) <- unbox bell00 ();
    (x,y) <- unbox alice (q,a);
    unbox bob (x,y,b)
```

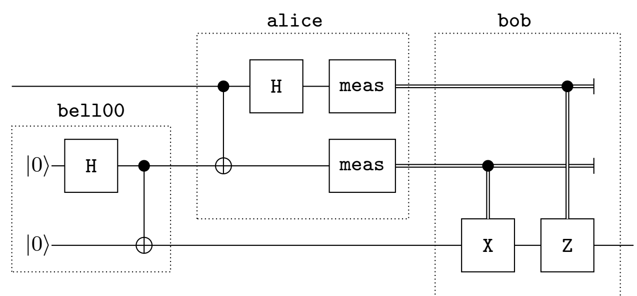
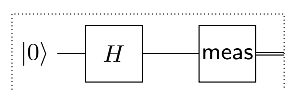


Figure 1. A $\mathcal{Q}$WIRE implementation of quantum teleportation.

```
flip : Bool =
  run (q <- gate init0 ();
       q <- gate H q;
       b <- gate meas q;
       output b)
```



3. The $\mathcal{Q}$WIRE Circuit Language

This section introduces the syntax and type theory of $\mathcal{Q}$WIRE and the interface for integrating $\mathcal{Q}$WIRE circuits into a host language.

3.1 Circuit language

As shown above, a circuit can be thought of as a sequence of gates on wires. These wires are described by their *wire type* W, which is either unit (has no data), a bit or qubit, or a tuple of wire types.[2]

$$W ::= 1 \mid \mathsf{bit} \mid \mathsf{qubit} \mid W_1 \otimes W_2$$

[2] Strictly speaking, the collection of wire types, along with the patterns for each wire type, could be thought of as an input to the system, provided that typing judgments for patterns are all syntax-directed. For example, we could consider a system without bit-valued wires, where measurement is only done via dynamic lifting. Alternatively we could consider more complex quantum data types in the style of Quipper (Green et al. 2013a). All wire types should be finite, however; see the discussion in Section 7.4 for more.

$\mathcal{Q}$WIRE is parameterized by a collection of gates $\mathcal{G}$, which each come equipped with input and output types. We write $\mathcal{G}(W_1, W_2)$ for the set of gates with input W_1 and output W_2. The gate set could consist of any collection of gates, but in the setting of quantum circuits it is conventional to choose a universal subset $\mathcal{U} \subseteq \mathcal{G}$ of unitary gates such that, for every $u \in \mathcal{U}(W, W)$, we also have

$$\begin{aligned} u^\dagger &\in \mathcal{U}(W, W) \\ \mathsf{control}\ u &\in \mathcal{U}(\mathsf{qubit} \otimes W, \mathsf{qubit} \otimes W) \\ \mathsf{bit\text{-}control}\ u &\in \mathcal{U}(\mathsf{bit} \otimes W, \mathsf{bit} \otimes W) \end{aligned}$$

Additionally, for the sake of this paper we assume we have initialization gates for bits and qubits:

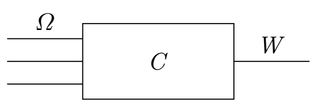
$$\mathsf{new0}, \mathsf{new1} \in \mathcal{G}(1, \mathsf{bit}), \mathsf{init0}, \mathsf{init1} \in \mathcal{G}(1, \mathsf{qubit})$$

as well as a measurement gate $\mathsf{meas} \in \mathcal{G}(\mathsf{qubit}, \mathsf{bit})$ and a discard gate $\mathsf{discard} \in \mathcal{G}(\mathsf{bit}, 1)$ for bits.

A typing judgment $\Gamma; \Omega \vdash C : W$ specifies when a circuit is well-formed. In this judgment,

- C is a circuit;
- $\Omega = w_1 : W_1, \ldots, w_n : W_n$ is a context of input wire names with their wire types;
- $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a context of host language variables with their host language types; and
- W is the output type of the circuit.

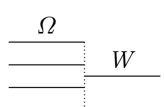
Thus, all well-typed circuits have the following shape:



Wires in $\mathcal{Q}$WIRE are *linear*, which means that they cannot be duplicated or discarded,[3] and when we write Ω, Ω' we assume that Ω and Ω' contain only disjoint wire names. Both Ω and Γ are thought of as *unordered* contexts.

The output of a circuit is built up as a *pattern* of its input wires:

$$\frac{\Omega \Rightarrow p : W}{\Gamma; \Omega \vdash \mathsf{output}\ p : W}$$

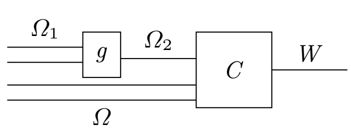


A pattern is just a tuple of wires identifying a single wire type.

$$\frac{}{\cdot \Rightarrow () : 1} \qquad \frac{}{w : W \Rightarrow w : W} \qquad \frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2}{\Omega_1, \Omega_2 \Rightarrow (p_1, p_2) : W_1 \otimes W_2}$$

A gate can be applied to a pattern of wires when permitted by the signature of the gate. The output of that gate is then decomposed by another pattern. The wires exiting the gate can then be used in the remainder of the circuit.

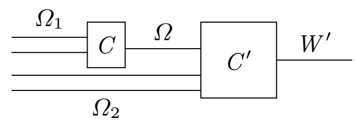
$$\frac{\begin{array}{l} g \in \mathcal{G}(W_1, W_2) \\ \Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad \Gamma; \Omega_2, \Omega \vdash C : W \end{array}}{\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1 ; C : W}$$



[3] Of course, gates may exist that duplicate or discard bits, but wires themselves are linear structures.

We compose circuits by connecting the output of one circuit to the input wires of another. This operation differs from sequential composition in that the second circuit may have additional inputs.

$$\frac{\Gamma; \Omega_1 \vdash C : W \quad \Omega \Rightarrow p : W \quad \Gamma; \Omega, \Omega_2 \vdash C' : W'}{\Gamma; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'}$$



3.2 Host language

In the QRAM model, a classical computer works together with a quantum computer. The classical computer communicates with the quantum computer by sending it instructions—that is, circuits in $\mathcal{Q}$WIRE. Terms in the host language, meanwhile, describe computations on the classical computer. We refer to the host language as HOST and describe some of its properties.

We assume that HOST is statically typed, and write its types as A. Furthermore, we assume that for each wire type there is a corresponding classical type—for example, a host-level boolean might correspond to the qubit and bit wire types, and tensor wire types correspond to pairs. In addition, we add a type representing the $\mathcal{Q}$WIRE circuits between two wire types, which we write $\mathsf{Circ}(W_1, W_2)$. Of course, HOST will often contain many other types, including functions and inductive data types, but the interface with $\mathcal{Q}$WIRE does not depend on the particular structure of HOST. For this reason we say that HOST is arbitrary: many different languages could fill in for the host language of $\mathcal{Q}$WIRE.

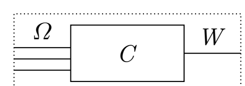
Overall, we can summarize the types of HOST as follows:

$$A ::= \cdots \mid \mathsf{Unit} \mid \mathsf{Bool} \mid A \times A \mid \mathsf{Circ}(W_1, W_2)$$

The typing judgment for HOST terms is written $\Gamma \vdash t : A$ where Γ is a context of variables with their associated types.

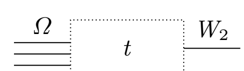
Boxing and Unboxing. The Circ type bridges $\mathcal{Q}$WIRE circuits and HOST terms. The type $\mathsf{Circ}(W_1, W_2)$ is a wrapper around $\mathcal{Q}$WIRE circuits of the form $\Gamma; \Omega \vdash C : W_2$, where the wires in Ω come from a pattern destructing the input type W_1.

$$\frac{\Omega \Rightarrow p : W_1 \quad \Gamma; \Omega \vdash C : W_2}{\Gamma \vdash \mathsf{box}\ (p : W_1) \Rightarrow C : \mathsf{Circ}(W_1, W_2)}$$



A boxed term of type $\mathsf{Circ}(W_1, W_2)$ can be coerced back into a $\mathcal{Q}$WIRE circuit by describing how to match up the available input wires to the input type of the boxed representation.

$$\frac{\Gamma \vdash t : \mathsf{Circ}(W_1, W_2) \quad \Omega \Rightarrow p : W_1}{\Gamma; \Omega \vdash \mathsf{unbox}\ t\ p : W_2}$$



Lifting. In the QRAM model described above, the quantum computer also communicates with the classical computer by sending it the results of measurement. For example, given a circuit with no input wires and a bit output, *running* that circuit should result in a host language boolean value.

$$\frac{\Gamma; \cdot \vdash C : \mathsf{bit}}{\Gamma \vdash \mathsf{run}\ C : \mathsf{Bool}}$$

We can generalize this operation so that running a circuit that outputs a qubit implicitly measures that qubit and returns the corresponding boolean. In fact this relationship generalizes to any wire type, which can be *lifted* to a classical type as follows:

$$\begin{aligned} |\mathsf{bit}| &= \mathsf{Bool} & |1| &= \mathsf{Unit} \\ |\mathsf{qubit}| &= \mathsf{Bool} & |W_1 \otimes W_2| &= |W_1| \times |W_2| \end{aligned}$$

The *run* operator now has the following form:

$$\frac{\Gamma; \cdot \vdash C : W}{\Gamma \vdash \mathsf{run}\ C : |W|}$$

Run is a *static lifting* operator, meaning that there is no residual state left on the quantum computer after $\mathsf{run}\ C$ has completed. In contrast, *dynamic lifting* describes the case when, over the course of a quantum computation, a subset of the wires are measured and communicated to the classical computer. In this case, the classical computer uses those results to compute the remainder of the quantum circuit, and eventually sends the results to the quantum computer. Dynamic lifting is expensive because while the classical computer is computing the rest of the circuit, the existing state on the quantum computer must continuously undergo error correction to prevent degradation. However, dynamic lifting is a fundamental form of communication between the two machines, and is needed to implement algorithms like quantum error correction.

We write the dynamic lifting operator $x \Leftarrow \mathsf{lift}\ p; C$ to mean that the wires in p are measured, lifted to the classical computer as the host variable x, and used to compute the circuit C.

$$\frac{\Omega \Rightarrow p : W \quad \Gamma, x : |W|; \Omega' \vdash C : W'}{\Gamma; \Omega, \Omega' \vdash x \Leftarrow \mathsf{lift}\ p; C : W'}$$

The dynamic and static lifting operations are not mutually derivable, as they represent two fundamentally different ways to communicate the results of measurement between the two systems.

3.3 Static semantics

To summarize, the syntax of $\mathcal{Q}$WIRE circuits and HOST terms include the following:

$$\begin{array}{lrl} \text{(Patterns)} & p ::= & () \mid w \mid (p, p) \\ \text{(Circuits)} & C ::= & \mathsf{output}\ p \mid p_2 \leftarrow \mathsf{gate}\ g\ p_1; C \mid p \leftarrow C; C \\ & & \mid x \Leftarrow \mathsf{lift}\ p; C \mid \mathsf{unbox}\ t\ p \\ \text{(Terms)} & t ::= & \cdots \mid \mathsf{run}\ C \mid \mathsf{box}\ (p : W) \Rightarrow C \end{array}$$

The typing rules are summarized in Figure 2. Note that we often write $\mathsf{box}\ p \Rightarrow C$ instead of $\mathsf{box}\ (p : W) \Rightarrow C$ when the type of the input pattern is clear. Note that typing contexts are unique for both patterns and circuits:

Lemma 1. *If $\Omega_1 \Rightarrow p : W$ and $\Omega_2 \Rightarrow p : W$ then $\Omega_1 = \Omega_2$. If $\Gamma; \Omega_1 \vdash C : W$ and $\Gamma; \Omega_2 \vdash C : W$ then $\Omega_1 = \Omega_2$.*

4. Operational semantics: circuit normalization

Circuits in $\mathcal{Q}$WIRE represent instructions to be executed on a quantum computer: either apply a particular gate, or request a dynamic lifting operation. Composition and unbox operations are more like meta-operations: they describe ways to construct more complex combinations of gates. In this section we define an operational semantics that eliminates all instances of unboxing and composition, resulting in a small set of normal forms. The subset of $\mathcal{Q}$WIRE circuits in normal forms are identified by two main properties.

First, normal circuits should operate only on bits and qubits, not on the tuples of wires described by arbitrary wire types W. We call a circuit *concrete* when all of its input wires are either bits or qubits:

$$\cdot; \mathcal{Q} \vdash C : W \qquad \text{where} \qquad \mathcal{Q} ::= \cdot \mid \mathcal{Q}, w : \mathsf{bit} \mid \mathcal{Q}, w : \mathsf{qubit}.$$

A concrete circuit is called *normal* when it consists only of gate applications, outputs, and dynamic lifting operations.

$$N ::= \mathsf{output}\ p \mid p_2 \leftarrow \mathsf{gate}\ g\ p_1; N \mid x \Leftarrow \mathsf{lift}\ p; C$$

$$\frac{\Omega \Rightarrow p : W}{\Gamma; \Omega \vdash \mathsf{output}\ p : W}\ \text{OUTPUT}$$

$$\frac{\begin{array}{l} g \in \mathcal{G}(W_1, W_2) \\ \Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad \Gamma; \Omega_2, \Omega \vdash C : W \end{array}}{\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1; C : W}\ \text{GATE}$$

$$\frac{\Gamma; \Omega_1 \vdash C : W \quad \Omega \Rightarrow p : W \quad \Gamma; \Omega, \Omega_2 \vdash C' : W'}{\Gamma; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'}\ \text{COMPOSE}$$

$$\frac{\Omega \Rightarrow p : W \quad \Gamma, x : |W|; \Omega' \vdash C : W'}{\Gamma; \Omega, \Omega' \vdash x \Leftarrow \mathsf{lift}\ p; C : W'}\ \text{LIFT}$$

$$\frac{\Gamma \vdash t : \mathsf{Circ}(W_1, W_2) \quad \Omega \Rightarrow p : W_1}{\Gamma; \Omega \vdash \mathsf{unbox}\ t\ p : W_2}\ \text{UNBOX}$$

$$\frac{\Gamma; \cdot \vdash C : W}{\Gamma \vdash \mathsf{run}\ C : |W|}\ \text{RUN}$$

$$\frac{\Omega \Rightarrow p : W_1 \quad \Gamma; \Omega \vdash C : W_2}{\Gamma \vdash \mathsf{box}\ (p : W_1) \Rightarrow C : \mathsf{Circ}(W_1, W_2)}\ \text{BOX}$$

Figure 2. Typing rules for $\mathcal{Q}$WIRE.

Notice that the lifting operator $x \Leftarrow \mathsf{lift}\ p; C$ does not assume that its continuation C is also normal. This is because C has a free host-level variable x that cannot in general be normalized. For example, consider the circuit $x \Leftarrow \mathsf{lift}\ w; \mathsf{unbox}\ (\mathtt{init}\ x)\ ()$: the continuation $\mathsf{unbox}\ (\mathtt{init}\ x)\ ()$ cannot be normalized because $\mathtt{init}\ x$ does not reduce in the host language.

In the rest of this section we define the small-step operational semantics that reduces concrete circuits typed by $\cdot; \mathcal{Q} \vdash C : W$ to normal circuits. The operational rules rely on a fairly complex substitution relation, which we briefly address.

Substitution. A substitution $\{p'/p\}$ describes a finite map from wire names to patterns. It is well-defined only when p generalizes p' (written $p' \preccurlyeq p$) in the following sense:

$$\frac{}{p' \preccurlyeq w} \qquad \frac{}{() \preccurlyeq ()} \qquad \frac{p_1' \preccurlyeq p_1 \quad p_2' \preccurlyeq p_2}{(p_1', p_2') \preccurlyeq (p_1, p_2)}$$

We say $p' \prec p$ when $p' \preccurlyeq p$ and $\neg(p \preccurlyeq p')$, and we say p is concrete for W when, for all $\Omega \Rightarrow p' : W$, $\neg(p' \prec p)$.

Lemma 2. *If $\Omega \Rightarrow p : W$ and $\mathcal{Q} \Rightarrow p' : W$, then $p' \preccurlyeq p$.*

The substitution map is defined as follows:

$$\begin{aligned} \{()/()\} &= \emptyset \\ \{p'/w\} &= w \mapsto p' \\ \{(p_1', p_2')/(p_1, p_2)\} &= \{p_1'/p_1\}, \{p_2'/p_2\} \end{aligned}$$

A well-defined substitution extends to *total* functions on patterns, circuits, and wire contexts. For patterns, we have:

$$\begin{aligned} ()\{p'/p\} &= () \\ w\{p'/p\} &= \begin{cases} p_0 & \text{if } w \mapsto p_0 \in \{p'/p\} \\ w & \text{otherwise} \end{cases} \\ (p_1, p_2)\{p'/p\} &= (p_1\{p'/p\}, p_2\{p'/p\}) \end{aligned}$$

The operation on circuits is straightforward, assuming the usual notions of capture-avoidance.

$$(\mathsf{output}\ p_0)\ \{p'/p\} = \mathsf{output}\ (p_0\{p'/p\})$$
$$(p_2 \leftarrow \mathsf{gate}\ g\ p_1; C)\ \{p'/p\} = p_2 \leftarrow \mathsf{gate}\ g\ p_1\{p'/p\}; C\ \{p'/p\}$$
$$(x \Leftarrow \mathsf{lift}\ p_0; C)\ \{p'/p\} = x \Leftarrow \mathsf{lift}\ p_0\{p'/p\}; C\ \{p'/p\}$$
$$(\mathsf{unbox}\ t\ p_0)\ \{p'/p\} = \mathsf{unbox}\ t\ (p_0\{p'/p\})$$
$$(p_0 \leftarrow C; C')\ \{p'/p\} = p_0 \leftarrow C\ \{p'/p\}; C'\ \{p'/p\}$$

A well-defined substitution $\{p'/p\}$ is *consistent with* w at W if $(w, p_0) \in \{p'/p\}$ implies that there is some (unique[4]) Ω_0 such that $\Omega_0 \Rightarrow p_0 : W$. A substitution is consistent with a context Ω when, for all $w : W \in \Omega$, it is consistent with w at W.

For wire contexts, suppose $\{p'/p\}$ is consistent with Ω. The substitution $\Omega\,\{p'/p\}$ is defined by induction on Ω:

$$\cdot\,\{p'/p\} = \cdot$$
$$(\Omega', w : W)\,\{p'/p\} = \begin{cases} \Omega'\,\{p'/p\}, \Omega_0 & \text{if } w \mapsto p_0 \in \{p'/p\} \\ & \text{and } \Omega_0 \Rightarrow p_0 : W \\ \Omega'\,\{p'/p\}, w : W & \text{otherwise} \end{cases}$$

Lemma 3. *Suppose $p' \preccurlyeq p$ where $\Omega \Rightarrow p : W$ and $\Omega' \Rightarrow p' : W$. Then:*

1. *If Ω'' is disjoint from Ω, then $\Omega''\,\{p'/p\} = \Omega''$.*
2. *$\Omega\,\{p'/p\} = \Omega'$.*
3. *$(\Omega_1, \Omega_2)\,\{p'/p\} = \Omega_1\,\{p'/p\}, \Omega_2\,\{p'/p\}$.*

Lemma 4. *Suppose $\{p'/p\}$ is consistent with Ω.*

1. *If $\Omega \Rightarrow p_0 : W$ then $\Omega\,\{p'/p\} \Rightarrow p_0\{p'/p\} : W$.*
2. *If $\Gamma; \Omega \vdash C : W$ then $\Gamma; \Omega\,\{p'/p\} \vdash C\ \{p'/p\} : W'$.*

Proof. Part 1 is immediate by induction. Part 2 is similarly by induction on the typing judgment $\Gamma; \Omega \vdash C : W$. The only difficult case concerns the bound patterns in gate and composition substitutions. For example, consider the gate application rule:

$$\frac{\begin{array}{l} g \in \mathcal{G}(W_1, W_2) \\ \Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad \Gamma; \Omega_2, \Omega \vdash C : W \end{array}}{\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1; C : W}$$

By part 1, we have $\Omega_1\,\{p'/p\} \Rightarrow p_1\{p'/p\} : W_1$, and by the inductive hypothesis we know $\Gamma; (\Omega_2, \Omega)\,\{p'/p\} \vdash C\ \{p'/p\} : W$. By α-equivalence, we can assume that the wires in Ω_2 are disjoint from the substitution $\{p'/p\}$, and so by Lemma 3, $(\Omega_2, \Omega)\,\{p'/p\} = \Omega_2, (\Omega\,\{p'/p\})$. Thus

$$\Gamma; \Omega_1\,\{p'/p\}, \Omega\,\{p'/p\} \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1\{p'/p\}; C\ \{p'/p\} : W. \quad \square$$

Operational Semantics. The small-step operational semantics for circuits is written $C \Longrightarrow C'$, and it depends on a similar operational semantics on terms, written $t \longrightarrow t'$. The relation on terms is made up of two parts, $\longrightarrow = \longrightarrow_{\mathrm{H}} \cup \longrightarrow_b$, where

1. $\longrightarrow_{\mathrm{H}}$ is the operational semantics derived from the host language alone, and
2. $\longrightarrow_b$ is the operational semantics for boxed circuits.

It is reasonable to assume that the host language relation $\longrightarrow_{\mathrm{H}}$ treats the type $\mathsf{Circ}(W_1, W_2)$ as an abstract data type, meaning that all terms of the form $\mathsf{box}\ p \Rightarrow C$ are treated as uninterpreted constants by the $\longrightarrow_{\mathrm{H}}$ relation. The relation $\longrightarrow_b$ reduces such a boxed circuit to one of the form $\mathsf{box}\ p' \Rightarrow N$ where p' is concrete for the type W_1. Let v refer to the values of HOST without

[4] Recall that Ω_0 is uniquely determined by the choice of p_0 and W (Lemma 1).

(Box)

$$\frac{p \text{ is concrete for } W \quad C \Longrightarrow C'}{\mathsf{box}\,(p : W) \Rightarrow C \longrightarrow_b \mathsf{box}\,(p : W) \Rightarrow C'}\ \textsc{struct}$$

$$\frac{p' \prec p \quad p' \text{ is concrete for } W}{(\mathsf{box}\,(p : W) \Rightarrow C) \longrightarrow_b (\mathsf{box}\ p' \Rightarrow C\ \{p'/p\})}\ \eta$$

(Unbox)

$$\frac{t \longrightarrow t'}{\mathsf{unbox}\ t\ p \Longrightarrow \mathsf{unbox}\ t'\ p}\ \textsc{struct}$$

$$\frac{}{\mathsf{unbox}\,(\mathsf{box}\,(p : W) \Rightarrow N)\ p' \Longrightarrow N\ \{p'/p\}}\ \beta$$

(Gate)

$$\frac{g \in \mathcal{G}(W_1, W_2) \quad p_2 \text{ is concrete for } W_2 \quad C \Longrightarrow C'}{p_2 \leftarrow \mathsf{gate}\ g\ p_1; C \Longrightarrow p_2 \leftarrow \mathsf{gate}\ g\ p_1; C'}\ \textsc{struct}$$

$$\frac{g \in \mathcal{G}(W_1, W_2) \quad p_2' \prec p_2 \quad p_2' \text{ is concrete for } W_2}{p_2 \leftarrow \mathsf{gate}\ g\ p_1; C \Longrightarrow p_2' \leftarrow \mathsf{gate}\ g\ p_1; C\ \{p_2'/p_2\}}\ \eta$$

(Composition)

$$\frac{C_1 \Longrightarrow C_1'}{p \leftarrow C_1; C_2 \Longrightarrow p \leftarrow C_1'; C_2}\ \textsc{struct}$$

$$\frac{}{p \leftarrow \mathsf{output}\ p'; C \Longrightarrow C\ \{p'/p\}}\ \beta$$

$$\frac{}{p \leftarrow (p_2 \leftarrow \mathsf{gate}\ g\ p_1; N); C \Longrightarrow p_2 \leftarrow \mathsf{gate}\ g\ p_1; p \leftarrow N; C}\ \mathrm{CC}$$

$$\frac{}{p' \leftarrow (x \Leftarrow \mathsf{lift}\ p; C'); C \Longrightarrow x \Leftarrow \mathsf{lift}\ p; p' \leftarrow C'; C}\ \mathrm{CC}$$

Figure 3. Operational semantics of concrete circuits.

circuits. Then v^{H} consists of values v along with boxed circuits as uninterpreted constants, and v^{C} consists of values along with normalized boxed circuits:

$$v^{\mathrm{H}} ::= v \mid \mathsf{box}\ p \Rightarrow C$$
$$v^{\mathrm{C}} ::= v \mid \mathsf{box}\ p \Rightarrow N$$

We explicitly do not describe the operational behavior of $\mathsf{run}\ C$ terms in this semantics. Instead, we assume that *run* operations reduce under $\longrightarrow_{\mathrm{H}}$; the host language has a facility to execute circuits on a (simulation of) a quantum computer in an appropriate way. Such an implementation is divorced from the *construction* of circuits, which is what we are developing in this section. One possibility is given in Section 5.1, where we give an example of a probabilistic operational rule for $\mathsf{run}\ C$ based on the denotational semantics of circuits.

The relations $\Longrightarrow$ on circuits and $\longrightarrow_b$ on boxed circuits are given in Figure 3. Each rule is labeled as either a structural rule (STRUCT), a β-reduction, an η-expansion, or a commuting conversion (CC).

The structural rules reduce circuits underneath binders. For composition and unboxing these structural rules are straightforward, in that they don't have any preconditions restricting when they apply. For boxes and gates, on the other hand, the continuations C of the circuit have some additional inputs that are not concrete even if the entire circuit is. For example, in the circuit $w \leftarrow \mathsf{gate}\ \mathsf{CNOT}\ (w_1, w_2); C$, the continuation C has a compound wire w even though the entire circuit has only concrete wires w_1 and w_2. To address this issue, the η-expansion rules for gates and boxes show that any such binding is equivalent to one with concrete inputs throughout.

Lemma 5. *If p is concrete for W then there is a unique $\mathcal{Q}$ such that $\mathcal{Q} \Rightarrow p : W$. Furthermore, for every wire type W there exists a p (not necessarily unique) such that p is concrete for W.*

Since an unbox operator is not a normal circuit, we eliminate it via a β rule once its argument t reaches a value of the form box $p \Rightarrow N$. Similarly, the composition operator reduces its first argument to a normal form before taking a step. When the argument is an output output p', the composition $p \leftarrow$ output $p'; C$ uses substitution to take a β-reduction step. On the other hand, when the argument consists of gate or lifting step, the semantics *commutes* that command to the front of the circuit; we call these operators *commuting conversions*.

4.1 Type safety.

We prove type safety with progress and preservation theorems, provided that the relation $\longrightarrow_{\mathrm{H}}$ is also type safe.

Theorem 6 (Preservation). *Suppose $\longrightarrow_{\mathrm{H}}$ satisfies preservation.*

1. *If $\vdash t : A$ and $t \longrightarrow t'$, then $\vdash t' : A$.*
2. *If $\cdot; \mathcal{Q} \vdash C : W$ and $C \Longrightarrow C'$, then $\cdot; \mathcal{Q} \vdash C' : W$.*

Proof. By induction on the step relation (**??**). □

Theorem 7 (Progress). *Suppose $\longrightarrow_{\mathrm{H}}$ satisfies progress with respect to the values v^{H}.*

1. *If $\cdot \vdash t : A$ then either t is a value v^{C} or there is some t' such that $t \longrightarrow t'$.*
2. *If $\cdot; \mathcal{Q} \vdash C : W$ then either C is normal or there is some C' such that $C \Longrightarrow C'$.*

Proof. By induction on the typing judgment (**??**). □

Provided that $\longrightarrow_{\mathrm{H}}$ is strongly normalizing, we can also show that circuits are strongly normalizing.

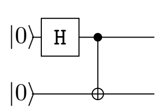
Theorem 8 (Normalization). *Suppose that $\longrightarrow_{\mathrm{H}}$ is strongly normalizing with respect to v^{H}.*

1. *If $\cdot \vdash t : A$, there exists some value v^{C} such that $t \longrightarrow^* v^{\mathrm{C}}$.*
2. *If $\cdot; \mathcal{Q} \vdash C : W$, there exists some normal circuit N such that $C \Longrightarrow^* N$.*

Proof. By induction on the number of constructors in the term and circuit (**??**). □

5. Denotational Semantics

In this section we will give a denotational semantics for $\mathcal{Q}$WIRE circuits. The state of a quantum system can be described in terms of a *density matrix*, in which numbers along the diagonal represent the probability of measuring a given state.[5] Consider, for instance, the entangled Bell pair produced by the following circuit:



[5] Formally, a density matrix is a positive Hermitian matrix whose trace sums to 1. Any pure state in column vector form can be transformed into a density matrix by taking its outer product with itself.

This pair of qubits is represented by the density matrix

$$\begin{pmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 \end{pmatrix}$$

where the $1/2$ in the top left represents the probability of measuring two zeros, while the $1/2$ in the bottom right represents the probability of measuring two ones. If we measured this system, we would obtain the *mixed state* density matrix

$$\begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 \end{pmatrix}$$

representing a distribution over $|00\rangle$ and $|11\rangle$.

Since a $\mathcal{Q}$WIRE circuit transforms some state to another, it will be interpreted as a superoperator over density matrices. In the rest of this section we will assume some familiarity with the mathematics of quantum computation; for reference we encourage readers to consult Nielsen and Chuang's standard text in the area (2010).

Given a Hilbert space $\mathcal{H}$, we write $\mathcal{H}^*$ for the collection of density matrices seen as linear transformations from $\mathcal{H}$ to $\mathcal{H}$. Given a linear map on Hilbert spaces $f : \mathcal{H} \multimap \mathcal{H}'$, f^* is a superoperator from $\mathcal{H}^*$ to $(\mathcal{H}')^*$ defined by $f^* \rho = f \rho f^\dagger$. In fact, every superoperator can be written

$$\Phi\, \rho = \sum_{i \in X} M_i^*\, \rho$$

for some indexed family of matrices $\{M_i\}_{iinX}$. We define

$$\left(\Phi \otimes \Phi'\right)\ \rho = \sum_{(i,j) \in X \times X'} (M_i \otimes M_j')^*\, \rho.$$

In this model, a wire type is interpreted as a Hilbert space in the following way:

$$[\mathsf{bit}] = \mathcal{H}_2 \qquad [1] = \mathcal{H}_1$$
$$[\mathsf{qubit}] = \mathcal{H}_2 \qquad [W_1 \otimes W_2] = [W_1] \otimes [W_2]$$

The intention is that a circuit from W_1 to W_2 is interpreted as a superoperator mapping density matrices corresponding to W_1 to density matrices corresponding to W_2. Notice that the denotation of bit and qubit are identical, which reflects the fact that bit-valued wires on a quantum machine are implemented using qubits in the $|0\rangle$ and $|1\rangle$ states.

For example, every gate $g \in \mathcal{G}(W_1, W_2)$ is be interpreted as a superoperator between W_1 and W_2. Although the set of gates is a parameter of the system, a unitary gate $\mathcal{U}$ should clearly correspond to $\mathcal{U}^*$, and the interpretation of other likely gates is as follows:

$$[\![\mathsf{new0}]\!], [\![\mathsf{init0}]\!] = (|0\rangle\, \langle 0|)^*$$
$$[\![\mathsf{new1}]\!], [\![\mathsf{init1}]\!] = (|1\rangle\, \langle 1|)^*$$
$$[\![\mathsf{meas}]\!] = (|0\rangle\, \langle 0|)^* + (|1\rangle\, \langle 1|)^*$$
$$[\![\mathsf{discard}]\!] = \langle 0|^* + \langle 1|^*$$

$\mathcal{Q}$WIRE circuits are specified by an unordered context of input wires Ω. However, we can equally well think of Ω as an *ordered* context, along with an explicit permutation rule to change the order of the wires.[6]

$$\frac{\Gamma; \Omega' \vdash C : W \quad \pi : \Omega \equiv \Omega'}{\Gamma; \Omega \vdash C : W}$$

[6] We elide these details in Section 3 as they complicate the operational semantics.

Permutations are defined inductively.

$$\frac{}{\epsilon : \Omega \equiv \Omega} \qquad \frac{\pi_1 : \Omega_1 \equiv \Omega_2 \quad \pi_2 : \Omega_2 \equiv \Omega_3}{\pi_2 \circ \pi_1 : \Omega_1 \equiv \Omega_3}$$

$$\frac{}{(\mathsf{swap}\ \Omega_1\ \Omega_2) : \Omega, \Omega_1, \Omega_2, \Omega' \equiv \Omega, \Omega_2, \Omega_1, \Omega'}$$

Note that permutations are reflected in the typing judgments of circuits but not in the syntax. We extend the substitution relation to permutations in a natural way, writing $\pi\,\{p'/p\}$.

$$\epsilon\,\{p'/p\} = \epsilon$$
$$(\pi_2 \circ \pi_1)\,\{p'/p\} = \pi_2\,\{p'/p\} \circ \pi_1\,\{p'/p\}$$
$$(\mathsf{swap}\ \Omega_1\ \Omega_2)\,\{p'/p\} = \mathsf{swap}\ (\Omega_1\,\{p'/p\})\ (\Omega_2\,\{p'/p\})$$

An ordered context of wires is now interpreted as a Hilbert space by treating the comma as the tensor product:

$$[\cdot] = \mathcal{H}_1 \qquad [w : W] = [W] \qquad [\Omega_1, \Omega_2] = [\Omega_1] \otimes [\Omega_2]$$

Although the context of wires can be permuted inside a circuit, it will not be permuted inside a pattern. Therefore, a pattern $\Omega \Rightarrow p : W$ is just a reassociation of the input wires; all permutations must be done outside the pattern. This means that whenever $\Omega \Rightarrow p : W$, it must be the case that $[\Omega] = [W]$.

A permutation $\pi : \Omega \equiv \Omega'$ will be interpreted as a linear isomorphism from $[\Omega]$ to $[\Omega']$, written $[\pi]$, as follows:

$$[\epsilon] = \mathbf{I}$$
$$[\pi_2 \circ \pi_1] = [\pi_2] \circ [\pi_1]$$
$$[\mathsf{swap}\ \Omega_1\ \Omega_2](v_0 \otimes v_1 \otimes v_2 \otimes v_3) = (v_0 \otimes v_2 \otimes v_1 \otimes v_3)$$

Lemma 9. *If $\pi : \Omega \equiv \Omega'$ and $\{p'/p\}$ is consistent with Ω, then $[\pi\,\{p'/p\}] = [\pi]$.*

Proof. Straightforward by induction on the permutation. □

For $\cdot \vdash v : |W|$, we define $[\![v : |W|]\!]$ to be an element of $[W]$:

$$[\![* : \mathsf{Unit}]\!] = |*\rangle$$
$$[\![\mathsf{false} : \mathsf{Bool}]\!] = |0\rangle$$
$$[\![\mathsf{true} : \mathsf{Bool}]\!] = |1\rangle$$
$$[\![(v_1, v_2) : |W_1| \times |W_2|]\!] = [\![v_1 : |W_1|]\!] \otimes [\![v_2 : |W_2|]\!]$$

Now, for $\cdot; \Omega \vdash C : W$, we write $[\![\Omega \vdash C : W]\!]$ for its interpretation as a superoperator between $[\Omega]^*$ and $[W]^*$. Furthermore, for $\cdot \vdash t : \mathsf{Circ}(W_1, W_2)$, we write $[\![t]\!]$ for $[\![\Omega \vdash C : W_2]\!]$ where $t \longrightarrow_{\mathrm{H}}^* \mathsf{box}\ p \Rightarrow C$ in the host language and $\Omega \Rightarrow p : W_1$. This operation is functional exactly when the host language semantics is strongly normalizing.

The interpretation of circuits is defined in Figure 4.

Lemma 10. *If $\cdot; \Omega \vdash C : W$ and $\{p'/p_0\}$ is consistent with Ω, then*

$$[\![\Omega\,\{p'/p\} \vdash C\,\{p'/p\} : W]\!] = [\![\Omega \vdash C : W]\!].$$

Proof. By induction on the typing judgment. The proof is almost completely straightforward because the interpretation of circuits does not depend on the content of patterns. □

Theorem 11 (Soundness)**.** *If $\cdot; \mathcal{Q} \vdash C : W$ and $C \Longrightarrow C'$, then*

$$[\![\mathcal{Q} \vdash C : W]\!] = [\![\mathcal{Q} \vdash C' : W]\!].$$

Proof. By induction on the typing judgment (**??**). □

5.1 Operational behavior of *run*

In Section 4 we left the semantics of the *run* operator up to the choice of implementation—to be executed as either a simulator or on an actual quantum computer. Given the denotational semantics described in this section, however, we specify the correctness of $\mathsf{run}\ C$ as a probabilistic operation. If $\cdot; \cdot \vdash C : W$, then

$$[\![\cdot \vdash C : W]\!] I_1$$

is a density matrix for $[W]$. The basis for $[W]$ is isomorphic to $\{[\![v_i : |W|]\!] \mid \cdot \vdash v_i : |W|\}$, corresponding to the values of $|W|$, so we can write the density matrix $[\![C]\!] I$ as

$$\begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & \ddots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nn} \end{pmatrix}$$

according to this basis. Then for each i, we say that the probability of C being v_i is $\alpha_{i\,i}$, written $\mathsf{prob}(C = v_i) = \alpha_{i\,i}$. The operational semantics rule for $\mathsf{run}\ C$ can be summarized with respect to this relation: $\mathsf{run}\ C$ steps to v_i with probability $\alpha_{i\,i}$.

$$\frac{\mathsf{prob}(C = v_i) = \alpha_{i\,i}}{\mathsf{run}\ C \longrightarrow^{\alpha_{i\,i}} v_i}$$

6. Extensions to HOST

In this section we consider two extensions to the host language that expand the expressivity of $\mathcal{Q}$WIRE.

6.1 Case analysis of circuits

Thanks to the operational semantics in Section 4, we know that every circuit $\vdash t : \mathsf{Circ}(W_1, W_2)$ normalizes to $\mathsf{box}\ p \Rightarrow N$, where $\mathcal{Q} \Rightarrow p : W_1$ and $\cdot; \mathcal{Q} \vdash N : W_2$ for some concrete context $\mathcal{Q}$. Intuitively, this means that circuits can be inspected and analyzed after they are created, by case analysis on the structure of N. In this section we develop the infrastructure needed to do this kind of case analysis on boxed circuits and illustrate a safe circuit reversal function, written directly in the host language.[7]

Consider a function that reverses a circuit if all of its gates are unitary:

$$\texttt{reverse} : \mathsf{Circ}(W_1, W_2) \to \mathsf{Option}\ \mathsf{Circ}(W_2, W_1).$$

A first attempt at reverse examines the structure of the normal circuit underneath the hood:

```
reverse x =
  case x of
  | (box p => output p') -> Some (box p' => output p)
  | (box p => gate p2 = g p1 in N') -> ?
  | (box p => lift x = p' in C) -> None
  end.
```

When the circuit is a gate application, we would like to do further case analysis on both the structure of N' and the gate g. However, N' is a $\mathcal{Q}$WIRE circuit, not a host-language term of the Circ type, so the recursive call would have to be on $\mathsf{box}\ p_0 \Rightarrow N'$ for some pattern p_0 whose value we don't know. More significantly, N' is not a host-level variable at all, it is firmly in the circuit language, as are the patterns p, p_1, and p_2, as well as the gate g.

In order to perform case analysis of circuits inside the host language, we need two things: a host-level representation of gates and patterns, and an inductive data structure that we can prove is equivalent to $\mathsf{Circ}(W_1, W_2)$.

[7] Circuit reversal is quite a common operation in quantum circuits. Existing quantum circuit languages provide `reverse` as a built-in operation that may fail at runtime if the circuit is not reversible (Green et al. 2013a; Wecker and Svore 2014).

$$\frac{\Omega \Rightarrow p : W}{\cdot; \Omega \vdash \mathsf{output}\ p : W} \qquad [\![\Omega \vdash \mathsf{output}\ p : W]\!] = \mathbf{I}^*$$

$$\frac{\cdot; \Omega' \vdash C : W \quad \pi : \Omega \equiv \Omega'}{\cdot; \Omega \vdash C : W} \qquad [\![\Omega \vdash C : W]\!] = [\![\Omega' \vdash C : W]\!] \circ [\pi]^*$$

$$\frac{\cdot \vdash t : \mathsf{Circ}(W_1, W_2) \quad \Omega \Rightarrow p : W_1}{\cdot; \Omega \vdash \mathsf{unbox}\ t\ p : W_2} \qquad [\![\Omega \vdash \mathsf{unbox}\ t\ p : W']\!] = [\![t : \mathsf{Circ}(W, W')]\!]$$

$$\frac{g \in \mathcal{G}(W_1, W_2) \quad \Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad \cdot; \Omega_2, \Omega \vdash C : W}{\cdot; \Omega_1, \Omega \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1; C : W} \qquad [\![\Omega_1, \Omega \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1; C : W]\!] = [\![\Omega_2, \Omega \vdash C : W]\!] \circ ([\![g]\!] \otimes \mathbf{I}^*)$$

$$\frac{\Omega \Rightarrow p : W \quad x : |W|; \Omega' \vdash C : W'}{\cdot; \Omega, \Omega' \vdash x \Leftarrow \mathsf{lift}\ p; C : W'} \qquad [\![\Omega, \Omega' \vdash x \Leftarrow \mathsf{lift}\ p; C : W']\!] = \sum_{\cdot \vdash v : |W|} [\![\Omega' \vdash C\{v/x\} : W']\!] \circ ([v : |W|]^\dagger \otimes \mathbf{I})^*$$

$$\frac{\cdot; \Omega_1 \vdash C : W \quad \Omega_0 \Rightarrow p : W \quad \cdot; \Omega_0, \Omega_2 \vdash C' : W'}{\cdot; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'} \qquad [\![\Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W']\!] = [\![\Omega_0, \Omega_2 \vdash C' : W']\!] \circ ([\![\Omega_1 \vdash C : W]\!] \otimes \mathbf{I}^*)$$

Figure 4. Denotational semantics of circuits.

Gates. A host-level representation of gates is straightforward:

$$\frac{g \in \mathcal{G}(W_1, W_2)}{\Gamma \vdash g : \mathsf{Gate}(W_1, W_2)}$$

We expect a small number of operations on host-level gates, e.g.

$\texttt{isUnitary} : \mathsf{Gate}(W_1, W_2) \to \mathsf{Bool}$
$\texttt{transpose} : \mathsf{Gate}(W_1, W_2) \to \mathsf{Option\ Gate}(W_2, W_1)$

Patterns. A host-level pattern can be constructed in a similar way to a host level circuit: we write $\mathsf{pat}\ p \Rightarrow p'$ and think of it as a function between wire types. Host-level patterns can also be unpacked in a way similar to unboxing:

$$\frac{\Gamma; \Omega \Rightarrow p_1 : W_1 \quad \Gamma; \Omega \Rightarrow p_2 : W_2}{\Gamma \vdash \mathsf{pat}\ (p_1 : W_1) \Rightarrow p_2 : \mathsf{Pat}(W_1, W_2)}$$

$$\frac{\Gamma \vdash t : \mathsf{Pat}(W_1, W_2) \quad \Gamma; \Omega \Rightarrow p : W_1}{\Gamma; \Omega \Rightarrow \mathsf{unpat}\ t\ p : W_2}$$

The addition of the $\mathsf{Pat}(W_1, W_2)$ type means a few things: the pattern typing judgment must include host-language variables, and patterns now normalize just like circuits do. In particular, normal patterns are any of the form

$$n ::= () \mid w \mid (n_1, n_2)$$

and unpacking patterns proceeds by the substitution we already defined in Section 4.

$$\mathsf{unpat}\ (\mathsf{pat}\ p_1 \Rightarrow p_2)\ p \Longrightarrow p_2\{p/p_1\}$$

Again, in order for substitution to be valid, it must be the case that the underlying pattern is concrete, for example:

$$\mathsf{pat}\ (p_1 : W) \Rightarrow p_2 \Longrightarrow \mathsf{pat}\ p_1' \Rightarrow p_2\{p_1'/p_2\}$$

when p_1' is concrete for W and $p_1' \prec p_1$.

The progress and preservation theorems for patterns fall out naturally from the substitution lemma (Lemma 4).

We can reverse a host-level pattern in the following way:

```
reverse_pat (p : Pat(W1,W2)) : Pat(W2,W1) =
  pat (unpat p w) => w
```

We can show that `reverse_pat (reverse_pat p) = p`. Suppose $p = (\mathsf{pat}\ p_1 \Rightarrow p_2)$ where both p_1 and p_2 are concrete. Then:

$$\begin{aligned} \texttt{reverse\_pat}\ p &= \mathsf{pat}\ (\mathsf{unpat}\ p\ w_1) \Rightarrow w_1 \\ &=_\eta \mathsf{pat}\ (\mathsf{unpat}\ p\ p_1) \Rightarrow p_1 \\ &=_\beta \mathsf{pat}\ (p_2\{p_1/p_1\}) \Rightarrow p_1 = (\mathsf{pat}\ p_2 \Rightarrow p_1). \end{aligned}$$

It follows immediately that `reverse_pat` $(\texttt{reverse\_pat}\ p) = \mathsf{pat}\ p_1 \Rightarrow p_2$.

Pattern Matching. Given the host-language representations of patterns and gates, we can start to axiomatize the structure of circuits in the host language. For example, an output circuit of type $\mathsf{Circ}(W_1, W_2)$ is represented by a host-level pattern $\mathsf{Pat}(W_1, W_2)$.

A gate application of $g : \mathsf{Gate}(W_1', W_2')$ consists first of a pattern $\mathsf{Pat}(W_1, W_1' \otimes W_0)$ breaking the input W_1 into two parts: the input to the gate W_1', and the unused wires W_0. The continuation of the circuit then has the type $\mathsf{Circ}(W_2' \otimes W_0, W_2)$.

A dynamic lifting operator similarly starts with a pattern $\mathsf{Pat}(W_1, W \otimes W')$ that breaks up the input into the part that will be measured and the continuation of the circuit. The continuation is represented as a function from the result of the lifting, $|W|$, to a circuit $\mathsf{Circ}(W', W_2)$.

Put another way, the type $\mathsf{Circ}(W_1, W_2)$ is isomorphic to the following indexed data type:

```
type ICirc W1 W2 =
| Output : Pat W1 W2 -> ICirc W1 W2
| Gate   : Pat W1 (W1'⊗W0) -> Gate W1' W2' ->
           Circ(W2'⊗W0, W2) -> ICirc W1 W2
| Lift   : Pat W1 (W⊗W') ->
           (|W| -> Circ(W',W2)) -> ICirc W1 W2.
```

We can write the function that embeds an inductive circuit into a $\mathcal{Q}$WIRE circuit directly in the host language:

```
fromICirc (t : ICirc W1 W2) : Circ(W1,W2) =
  case t of
  | Output p   -> box w1 => output (unpat p w1)
  | Gate p g c ->
    box (unpat (reverse_pat p) (w1,w0)) =>
      w2 <- gate g w1;
      unbox c (w2,w0)
  | Lift p f   ->
    box (unpat (reverse_pat p) (w,w')) =>
      x <- lift w;
      unbox (f x) w'
  end
```

The function from $\mathsf{Circ}(W_1, W_2)$ to $\mathsf{ICirc}\ W_1\ W_2$ is loosely the algorithm described above, and has the type signature

```
toICirc (t : Circ(W1,W2)) : ICirc W1 W2
```

However, `toICirc` is not expressible directly in the host language, since it relies on induction on the typing structure of circuits. Instead we describe it in the meta-theory. If $\mathcal{Q} \Rightarrow p : W_1$ and $\cdot; \mathcal{Q} \vdash N : W_2$ then `toICirc` $(\mathsf{box}\ p \Rightarrow N)$ is defined on the typing structure of N, as shown in Figure 5.

Theorem 12. *For all terms* `t` *of type* `ICirc W1 W2` *and* `c` *of type* $\mathsf{Circ}(W_1, W_2)$*, we have:*

$$\texttt{toICirc (fromICirc t)} = \texttt{t}$$
$$\texttt{fromICirc (toICirc c)} = \texttt{c}$$

Proof. By induction on the typing judgment (**??**). ☐

Reversing circuits. The circuit reversal function can be written by interfacing with the `ICirc` type.

```
reverse (c : Circ(W1,W2)) : Option (Circ(W2,W1)) =
  case toICirc c of
  | Output p -> fromICirc (Output (reverse_pat p))
  | Gate p g c' ->
    case reverse (toICirc c'), reverse_gate g of
    | Some c_rev, Some g_rev ->
      let p_rev = reverse_pat p in
      let i_rev = Gate id_pat g_rev (Output p_rev) in
      inSeq c_rev (fromICirc i_rev)
    | _, _ -> None
    end
  | Lift _ _ -> None
  end
```

where `id_pat = pat w => w` and `inSeq` is the sequential composition operator defined in Section 2. We also assume the existence of an operation `reverse_gate` on gates that is semantically valid, so that if `reverse_gate` $g = \mathsf{Some}\ g'$, then

$$[\![g]\!] \circ [\![g']\!] = \mathbf{I}^* = [\![g']\!] \circ [\![g]\!].$$

In that case, we can prove the following correctness condition:

Theorem 13. *If* `reverse c` $=$ `Some c'` *then*

$$[\![c]\!] \circ [\![c']\!] = \mathbf{I}^* \quad \textit{and} \quad [\![c']\!] \circ [\![c]\!] = \mathbf{I}^*.$$

Proof. By induction on terms (**??**). ☐

Other operations expressible in the host language with case analysis include:

- Less naive circuit reversal algorithms; for example qubit initialization can be reversed and treated as an ancilla if every operation following initialization can be reversed;
- Special purpose quantum simulators;
- A safe control operator on circuits that adds a control wire to every unitary gate and outputs None if it encounters a lift or non-unitary gate;
- Resource analyzers that count the number of gates in a circuit up to a dynamic lifting operation;
- An optimizer that collects the gates in a circuit into a data structure, runs an optimization pass, and reconstructs the circuit;
- A transformation that maps one set of unitary gates to another;
- A static analysis tool to determine whether two circuits are equivalent (Staton 2015).

Another way to gain expressivity of circuits is by adding dependent types to the host language.

6.2 Dependent types

Consider the quantum Fourier transform, which is a circuit with n inputs and n outputs. It is natural for the wire types of the Fourier circuit reflect this dependency on n. In the language of dependent types, it might have the signature

$$\texttt{fourier} : \Pi\,(n\!:\!\mathsf{Nat}).\mathsf{Circ}(\bigotimes n\ \mathsf{qubit}, \bigotimes n\ \mathsf{qubit})$$

where `tensor` is a type-level function that duplicates the argument wire type (qubit) some number of times (defined below).

Combining linear and dependent types is still an area of active research (Krishnaswami et al. 2015; McBride 2016) but thanks to the separation between the circuit and host languages, we can get away with a limited form of dependent types due to Krishnaswami et al. (2015). Under this strategy, types can depend on terms, but only terms of *classical* (non-linear) type. These include dependencies on wire types themselves, which are considered classical terms in the universe hierarchy.

To be more precise, let $\mathcal{W}$ be the kind of wire types, and consider an indexed hierarchy of host language types $\mathcal{A}_i$. We define the following well-formedness judgment: first, $\mathcal{W}$ has type $\mathcal{A}_i$ for any index i, and $\mathcal{A}_i$ has type $\mathcal{A}_{i+1}$:

$$\frac{}{\Gamma \vdash \mathcal{W} : \mathcal{A}_i} \qquad \frac{}{\Gamma \vdash \mathcal{A}_i : \mathcal{A}_{i+1}}$$

In addition, we introduce a new host-language type $\Pi\,(x : A).B$ with the following well-formedness condition:[8]

$$\frac{\Gamma \vdash A\!:\!\mathcal{A}_i \quad \Gamma, x\!:\!A \vdash B\!:\!\mathcal{A}_i}{\Gamma \vdash \Pi\,(x\!:\!A).B\!:\!\mathcal{A}_i}$$

Π types have the usual introduction and elimination rules:

$$\frac{\Gamma, x\!:\!A_1 \vdash t\!:\!A_2}{\Gamma \vdash \lambda x.t\!:\!\Pi\,(x\!:\!A_1).A_2} \qquad \frac{\Gamma \vdash t\!:\!\Pi\,(x\!:\!A_1).A_2 \quad \Gamma \vdash t'\!:\!A_1}{\Gamma \vdash t\,t'\!:\!A_2\{t'/x\}}$$

A more thorough analysis of this type structure is needed, but is beyond the scope of this paper.

A dependent QFT. Under this framework, we can start with the type-level function `tensor`:

```
tensor (n : Nat) (W : ℒ) : ℒ =
  case n of
  | 0 => 1
  | 1 => W
  | S n' => W ⊗ tensor n' W
  end
```

We write $\bigotimes n\ W$ for `tensor` $n\ W$.

Next we use these length-indexed tuples to write a dependently-typed quantum Fourier transform in the style of Green et al. (2013b). Our version of the Fourier circuit ensures that the number of qubits in the input and output are always the same.

First, we define the rotation circuits. We assume the presence of a family of gates `RGate m` that rotates its input along the z-axis by $\frac{2\pi i}{2^m}$ (Green et al. 2013b). The `rotations` circuit takes two natural number inputs: m, the argument given to the controlled R gates; and n, the number of bits in the input.

```
rotations (m:Nat) : Π (n:Nat).
  CIRC(⨂ (n+1) qubit, ⨂ (n+1) qubit) =
  fun n => case n of
  | 0    -> id
  | 1    -> id
  | S n' -> box (c,(q,qs)) =>
    (c,qs) <- unbox rotations m n' (c,qs);
    (c,q)  <- gate (control (RGate (2+m-n'))) (c,q);
    output (c,(q,w))
  end
```

[8] The presentation in this section is actually a simplification of the work of Krishnaswami et al. (2015), as we do not consider linear types with any dependencies.

$$\frac{\mathcal{Q} \Rightarrow p' : W_2}{\cdot; \mathcal{Q} \vdash \mathsf{output}\ p' : W_2} \qquad \texttt{toICirc}\ (\mathsf{box}\ p \Rightarrow \mathsf{output}\ p') = \texttt{Output}\ (\mathsf{pat}\ p \Rightarrow p')$$

$$\frac{\mathcal{Q}_1 \Rightarrow p_1 : W_1' \qquad \Omega_2 \Rightarrow p_2 : W_2' \qquad \cdot; \Omega_2, \mathcal{Q} \vdash N' : W_2}{\cdot; \mathcal{Q}_1, \mathcal{Q} \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1; N' : W_2} \qquad \texttt{toICirc}\ (\mathsf{box}\ p \Rightarrow p_2 \leftarrow \mathsf{gate}\ g\ p_1; N') = \texttt{Gate}\ (\mathsf{pat}\ p \Rightarrow (p_1, \overrightarrow{\mathcal{Q}}))\ g\ (\mathsf{box}\ (p_2, \overrightarrow{\mathcal{Q}}) \Rightarrow N')$$

$$\frac{\mathcal{Q}' \Rightarrow p' : W \qquad x : |W|; \mathcal{Q} \vdash C : W_2}{\cdot; \mathcal{Q}', \mathcal{Q} \vdash x \Leftarrow \mathsf{lift}\ p'; C : W_2} \qquad \texttt{toICirc}\ (\mathsf{box}\ p \Rightarrow x \Leftarrow \mathsf{lift}\ p'; C) = \texttt{Lift}\ (\mathsf{pat}\ p \Rightarrow (p', \overrightarrow{\mathcal{Q}}))\ (\lambda x. \mathsf{box}\ \overrightarrow{\mathcal{Q}} \Rightarrow C)$$

Figure 5. Definition of `toICirc` by induction on the structure of normal circuits. $\overrightarrow{\mathcal{Q}}$ is a canonical pattern made up from the wires in $\mathcal{Q}$.

The Fourier transform can now be defined in a type-safe way:

```
fourier : Π(n:Nat). CIRC(⊗ n qubit, ⊗ n qubit)=
  fun n => case n of
  | 0 => id
  | 1 => hadamard
  | S n' => box (q,w) =>
              w <- unbox fourier n' w;
              unbox rotations (S n') n' (q,w)
```

where `hadamard = box w => gate H w`.

7. Discussion

Thus far we have shown that $\mathcal{Q}$WIRE is a small, safe, and expressive circuit language. In the remainder of the paper we take a closer look at the similarities and differences between $\mathcal{Q}$WIRE and existing quantum circuit languages, with an eye towards future work.

7.1 The QRAM model

The driving design of $\mathcal{Q}$WIRE is the separation of classical computations in the host language from quantum computations in the circuit language. The inspiration for this model comes from two main sources.

On the logical side, $\mathcal{Q}$WIRE draws on Benton's (1995) linear/nonlinear logic (LNL), which partitions the exponential from Girard's linear logic (1987) into a purely linear fragment and a purely nonlinear fragment connected via a categorical adjunction. Variations on LNL have extended the logical framework to type systems for other substructural logics (Pfenning and Griffith 2015), polarized logics (Zeilberger 2008), and dependently-typed logics as in Section 6.2 (Krishnaswami et al. 2015).

On the quantum computing side, the QRAM model postulates a classical computer working alongside a quantum computer. QRAM is widely accepted as a programming model, although there is no clear consensus as to the degree to which the structure of quantum programming languages should reflect this separation.

At one end of the QRAM spectrum of language design is $\mathcal{Q}$WIRE, which syntactically separates quantum data inside circuits from classical data, and treats these two syntactic fragments as distinct languages. Bettelli et al.'s Q programming language (2003), takes a similar approach, treating circuits (called quantum operators) as an isolated subsystem inside a generic host language.

Quipper and LIQ$Ui|\rangle$ are based on the Quantum IO Monad (Altenkirch and Green 2010), which isolates quantum operations behind a monad. Indeed, the adjoint structure of $\mathcal{Q}$WIRE, when viewed from the host language, forms a similar monad, where the bind of the monad is implemented with dynamic lifting. However, unlike in $\mathcal{Q}$WIRE, qubits are first-class data in these systems, even though they cannot be constructed outside of the monad.

The separation between circuits and ordinary data has proved useful in the design of classical circuit languages as well. For example, in Haskell the *arrow* type class can be used to describe functional structures such as those corresponding to circuits (Hughes 2005). The fundamental constructor of arrows, which coerces a function in the host language to an arrow type, is not valid for $\mathcal{Q}$WIRE, although arrows have applications for non-circuit models of quantum computation (Vizzotto et al. 2009).

On the opposite end of the spectrum are languages like QML (Altenkirch and Grattage 2005), the quantum λ-calculus (Selinger and Valiron 2009), and QPL (Selinger 2004), which avoid dealing with circuits entirely by treating qubits as data. Having first-class qubits may lead to more natural programming abstractions, like partially applied higher-order functions or imperative loops. However it requires a much more involved type theory (for instance, linear subtyping in the quantum λ-calculus) to achieve type safety.

7.2 Type systems for well-formed circuits

$\mathcal{Q}$WIRE provides a type-safe circuit language within an arbitrary (type-safe) host language by keeping the circuit language minimal and pushing the remaining infrastructure to the host language. Embedded languages like Quipper and LIQ$Ui|\rangle$ do not cleanly separate embedded circuits from the host language, which means that verifying the embedded language requires verifying the combination host and circuit languages. For $\mathcal{Q}$WIRE we have shown that runtime errors in circuits can *only* arise from the host language, which is a maximal guarantee while still allowing arbitrary classical computations.

The type-safety guarantees gained from linear logic (e.g. respecting the no-cloning theorem) have been well-established by the quantum λ-calculus (Selinger and Valiron 2009). Quipper comes equipped with a programming idiom that recommends using quantum variables linearly except in certain circumstances, but programmers are unlikely to consistently follow this convention because it is neither enforced at compile time nor presented as a collection of unambiguous rules.

The Proto-Quipper project is an attempt to apply these foundations to a core language for Quipper with the goal of better runtime guarantees (Ross 2015). However, Proto-Quipper covers only a small subset of Quipper, and does not include measurement or initialization of qubits. Further, the classical component of Proto-Quipper is fixed, as it must be compatible with the underlying linear type system. Proto-Quipper is not a pure language, because its operational semantics imperatively constructs a circuit as the program runs and there is no equational theory. In contrast, the semantics of $\mathcal{Q}$WIRE is pure and equational reasoning is valid. Finally, the type system of Proto-Quipper makes extensive use of subtyping to account for linear use of quantum data. Although the type system makes it easier to write code without linearity annotations, it makes it harder to know when a term is well-typed. In $\mathcal{Q}$WIRE, the separation between the host language and circuit language makes linear typing easy and subtyping unnecessary.

An alternative to a linear type system is the Quantum IO Monad (Altenkirch and Green 2010). Although the monadic approach is sufficient to enforce no-cloning, by itself it is not strong enough to avoid all runtime errors. For example, Altenkirch and Green point out that extra *semantic conditions* based on the weakening property from linear logic are needed to safely type locally-bound ancilla and unitary conditional statements.

Although LIQ$Ui|\rangle$'s type system is loosely based on the Quantum IO monad, in LIQ$Ui|\rangle$ qubits and circuits are dynamically typed, and so certain operations, such as circuit reversal, may fail at runtime. Furthermore, LIQ$Ui|\rangle$ gates can always be applied to a list of qubits with the intention of operating on only a finite prefix of them. If the list is empty, any such operation could fail.

7.3 Denotational semantics and formal verification

Proto-Quipper has a type-safe operational semantics, but not a denotational semantics against which to compare. Conversely, LIQ$Ui|\rangle$ has a built-in denotational semantics since entangled qubits are represented directly by their *ket* state, which allows for the formal analysis of algorithms.

Although formal verification of algorithms is time-consuming, in the case of quantum computing the cost is likely worthwhile: quantum computing resources will be expensive for the foreseeable future, debugging is doubly difficult in a quantum setting, and testing using simulations is not scalable. Verification efforts related to LIQ$Ui|\rangle$ include an efficient compiler for a reversible fragment of the language in the formal theorem-prover F^* (Amy et al. 2016). Other verification projects based on denotational semantics for a variety of quantum languages exist on paper but not as machine-checked proofs for various simple quantum programming languages (D'Hondt and Panangaden 2006; Kakutani 2009; Ying 2011).

We expect $\mathcal{Q}$WIRE to be amenable to a similar kind of verification based on the denotational semantics presented in Section 5. In particular, we are interested in using a dependently-typed theorem prover like Coq (Coq Development Team 2015) as a host language, and using it to prove theorems about circuits. In fact, the dependently-typed infrastructure described in Section 6.2 was inspired by our investigations into a Coq implementation.

Verification based on equational theories of quantum computation (Staton 2015) is also well-suited for $\mathcal{Q}$WIRE. These equational theories characterize the semantic equivalence of circuits, such as the fact that `inSeq H H` is equivalent to the identity circuit. Such a theory could justify circuit optimizations provide a syntactic framework for program verification.

7.4 Usability

As a core circuit language, $\mathcal{Q}$WIRE is still missing many of the advanced features provided by Quipper and LIQ$Ui|\rangle$. As we look towards implementations of $\mathcal{Q}$WIRE in various host languages, we can learn from the features of more mature languages.

Parametric operators on circuits. Quipper and LIQ$Ui|\rangle$ both provide operations that globally transform circuits, including circuit reversal replacing one universal gate set with another, and applying optimizations. In general these operations are built into the language, and may fail at runtime if various conditions are not met. In $\mathcal{Q}$WIRE we have already illustrated how these operations can be written directly in the host language by (safely) extending it with a case analysis operation on circuits.

Automatic generation of quantum oracles. Quipper's quantum oracle feature uses Template Haskell (Sheard and Jones 2002) to generate a quantum circuit from an arbitrary classical function. By using Haskell as a host language we can imagine a similar extension to $\mathcal{Q}$WIRE.

Scalability. Quipper and LIQ$Ui|\rangle$ have both been used to successfully implement many nontrivial quantum algorithms (Siddiqui et al. 2014; Green et al. 2013a; Wecker and Svore 2014), in which the size of quantum circuits can grow into the millions of gates. One approach to scalability, embraced by LIQ$Ui|\rangle$, involves aggressive optimization and simulation, and is compatible with $\mathcal{Q}$WIRE using circuit case analysis. Another approach is to represent some circuits as black boxes when they are to be reused many times, recording their definition only once and (for example) precomputing their simulated behavior. This feature could be integrated into $\mathcal{Q}$WIRE by means of a function of type $\mathsf{Circ}(W_1, W_2) \to \mathsf{Gate}(W_1, W_2)$ that coerces boxed circuits into host-level gates.

Quantum data types. A quantum data type is any data type consisting of qubits, which is useful for describing modules like the quantum integers. Quipper provides a typeclass-based approach to quantum data types consisting of a data type of qubits along with a corresponding classical data type of booleans (corresponding to the lifted type $|W|$ in the syntax of $\mathcal{Q}$WIRE). In this paper we only consider tuples, but an extended system could easily allow other finite data types. Infinite data types are more problematic—in Quipper, infinite data types like lists must be *instantiated* at a finite size before generating circuits for them. A better solution is to include finitely *indexed* data types, such as the n-ary tuples of qubits shown in Section 6.2. Instantiation is enforced by the fact that $\Pi\,(x : \mathsf{Nat}).\mathsf{Circ}(\bigotimes x\ \mathsf{qubit}, \bigotimes x\ \mathsf{qubit})$ is not itself a circuit; it is a family of circuits that can be instantiated by feeding it a concrete natural number.

7.5 Conclusion

$\mathcal{Q}$WIRE is a minimal and highly modular core circuit language. It is minimal in that $\mathcal{Q}$WIRE has only five distinct commands, two of which are eliminated in the normalization procedure. It is modular in that $\mathcal{Q}$WIRE isn't attached to any specific programming language. We expect that the $\mathcal{Q}$WIRE interface will be useful in dependently-typed host languages like Coq for verification and formal analysis of circuits, in higher-order functional languages like Haskell, OCaml or F#, or potentially even in imperative languages like Python, Java, or C.

$\mathcal{Q}$WIRE uses linear types to enforce no-cloning, but does not allow them to spill over into the host language. This is crucial because linear types are the most natural way to enforce no-cloning, but are tremendously difficult to integrate into existing languages. $\mathcal{Q}$WIRE gets the best of both worlds by ensuring that circuits are linearly typed while allowing an arbitrarily powerful type system in the classical host language.

As a circuit description language, $\mathcal{Q}$WIRE is a low-level piece in the development of sophisticated quantum programming languages. Ultimately however, all quantum computation will boil down to circuit generation, necessitating the use of a circuit language like $\mathcal{Q}$WIRE. Having $\mathcal{Q}$WIRE as a safe, small circuit language is an excellent building block on which to rest the complex world of quantum computation.

Acknowledgments

We are grateful to Peter Selinger for his insights into quantum programming languages. This work is supported in part by the ONR MURI No. FA9550-16-1-0082, and by NSF Grants No. CCF-1421193 and DGE-1321851.

References

T. Altenkirch and J. Grattage. A functional quantum programming language. In *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, pages 249–258. IEEE, 2005.

T. Altenkirch and A. S. Green. The quantum IO monad. *Semantic Techniques in Quantum Computation*, pages 173–205, 2010.

M. Amy, M. Roetteler, and K. M. Svore. Verified compilation of space-efficient reversible circuits. Technical Report MSR-TR-2016-22, Microsoft Research, March 2016. URL `https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/`.

C. Badescu and P. Panangaden. Quantum alternation: Prospects and problems. In *Proceedings 12th International Workshop on Quantum Physics and Logic, QPL 2015, Oxford, UK, July 15-17, 2015.*, pages 33–42, 2015. doi: 10.4204/EPTCS.195.3.

P. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin Heidelberg, 1995. doi: 10.1007/BFb0022251.

S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *The European Physical Journal D*, 25(2):181–200, 2003.

Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.4*. 2015. Electronic resource, available from `http://coq.inria.fr`.

E. D'Hondt and P. Panangaden. Quantum weakest preconditions. *Mathematical Structures in Computer Science*, 16(03):429–451, 2006.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

A. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013, pages 333–342, 2013a.

A. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. An introduction to quantum programming in Quipper. In *Proceedings of the 5th International Conference on Reversible Computation*, volume 7948 of *Lecture Notes in Computer Science*, pages 110–124, 2013b. ISBN 978-3-642-38985-6.

J. Hughes. *Programming with Arrows*, pages 73–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi: 10.1007/11546382_2.

Y. Kakutani. A logic for formal verification of quantum programs. In *Advances in Computer Science-ASIAN 2009. Information Security and Privacy*, pages 79–93. Springer, 2009.

E. H. Knill. Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.

N. R. Krishnaswami, P. Pradic, and N. Benton. Integrating linear and dependent types. *SIGPLAN Notices*, 50(1):17–30, Jan. 2015. doi: 10.1145/2775051.2676969.

C. McBride. *I Got Plenty o' Nuttin'*, pages 207–233. Springer International Publishing, 2016. doi: 10.1007/978-3-319-30936-1_12.

M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

F. Pfenning and D. Griffith. Polarized substructural session types. In A. Pitts, editor, *Foundations of Software Science and Computation Structures*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-46678-0_1.

N. J. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie University, 2015.

P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, Aug. 2004.

P. Selinger and B. Valiron. Quantum lambda calculus. In S. Gay and I. Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University Press, 2009.

T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM. doi: 10.1145/581690.581691.

S. Siddiqui, M. J. Islam, and O. Shehab. Five quantum algorithms using Quipper. *arXiv preprint arXiv:1406.4481*, 2014.

S. Staton. Algebraic effects, linearity, and quantum programming languages. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 395–406, New York, NY, USA, 2015. ACM. doi: 10.1145/2676726.2676999.

J. K. Vizzotto, A. R. Du Bois, and A. Sabry. The arrow calculus as a quantum programming language. In H. Ono, M. Kanazawa, and R. de Queiroz, editors, *Proccedings of Logic, Language, Information and Computation: 16th International Workshop, WoLLIC 2009*, pages 379–393. Springer Berlin Heidelberg, June 2009. doi: 10.1007/978-3-642-02261-6_30.

D. Wecker and K. M. Svore. LIQUi|>: A software design architecture and domain-specific language for quantum computing. *arXiv:1402.4467 [quant-ph]*, 2014.

M. Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19, 2011.

M. Ying. Quantum recursion and second quantisation. May 2014. arXiv:1405.4443 [quant-ph].

N. Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3):66–96, 2008. doi: 10.1016/j.apal.2008.01.001. Special Issue: Classical Logic and Computation (2006).